c路由器转发实验实验报告

- 实验内容：

  完成对 ip.c,ip_base.c,icmp.c,arp.c,arpcache.c 的封装

  实验的大部分内容都是 分配空间 -> 赋值 -> 发包 展开的。

  首先考虑 arp.c

```
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    fprintf(stderr, "TODO: send arp request when lookup failed in
arpcache.\n");
    struct ether_header *arp_ether_header;
    struct ether_arp *arp_ether_arp;
    char *arp_packet;
    arp_packet = (char *)malloc(sizeof(struct ether_header) +sizeof(struct
ether_arp));
    arp_ether_header = (struct ether_header *)arp_packet;
    arp_ether_arp = (struct ether_arp*)(arp_packet + sizeof(struct
ether_header));
    for(int i = 0; i < ETH_ALEN; i++){
        arp_ether_header -> ether_dhost[i] = 0xff;
        arp_ether_header -> ether_shost[i] = iface->mac[i];
    }
    arp_ether_header -> ether_type = htons(ETH_P_ARP);
/*首先需要封装 ether头部和 arp头部，可以先申请一整块两者之和的空间，然后分别对其进行类型转
换。对于 ether头部的内容，需要对 dhost 和 shost 进行赋值，ether_type 的类型需要进行本
地字节序到网络字节序的转换。*/
    arp_ether_arp -> arp_hrd = htons(0X01);
    arp_ether_arp->arp_op = htons(0x01);
    arp_ether_arp -> arp_pro = htons(0x0800);
    arp_ether_arp -> arp_hln = 6;
    arp_ether_arp -> arp_pln = 4;
    for(int i = 0; i < ETH_ALEN; i++){
        arp_ether_arp -> arp_sha[i] = iface->mac[i];
        arp_ether_arp -> arp_tha[i] = 0x0;
    }
    arp_ether_arp -> arp_spa = htonl(iface->ip);
    arp_ether_arp -> arp_tpa = htonl(dst_ip);
    iface_send_packet(iface,  arp_packet , sizeof(struct ether_header) +
sizeof(struct ether_arp));
}
```

这一部分的主要需要注意对 网络字节序和本地字节序的转换。

```
// send an arp reply packet: encapsulate an arp reply packet, send it out
// through iface_send_packet
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)
{
    fprintf(stderr, "TODO: send arp reply when receiving arp request.\n");
    if(iface->ip == ntohl(req_hdr-> arp_tpa)){
        struct ether_header *arp_ether_header;
        struct ether_arp *arp_ether_arp;
```

```
        char *arp_packet;
        arp_packet = (char *)malloc(sizeof(struct ether_header) +
sizeof(struct ether_arp));
        arp_ether_header = (struct ether_header *)arp_packet;
        arp_ether_arp = (struct ether_arp *)(arp_packet + sizeof(struct
ether_header));
        for(int i = 0; i < ETH_ALEN; i++){
            arp_ether_header -> ether_dhost[i] = req_hdr-> arp_sha[i];
            arp_ether_header -> ether_shost[i] = iface-> mac[i];
        }
        arp_ether_header -> ether_type = htons(ETH_P_ARP);

        arp_ether_arp -> arp_hrd = htons(0x01);
        arp_ether_arp -> arp_op = htons(0x02);
        arp_ether_arp -> arp_pro = htons(0x0800);
        arp_ether_arp -> arp_hln = 6;
        arp_ether_arp -> arp_pln = 4;
        for(int i = 0; i < ETH_ALEN; i++){
            arp_ether_arp -> arp_sha[i] = iface-> mac[i];
            arp_ether_arp -> arp_tha[i] = req_hdr-> arp_sha[i];
        }
        arp_ether_arp -> arp_spa = htonl(iface->ip);
        arp_ether_arp -> arp_tpa = req_hdr -> arp_spa;
        iface_send_packet(iface, arp_packet, sizeof(struct ether_header) +
sizeof(struct ether_arp));
    }
}
```

和上面的基本类似。

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    fprintf(stderr, "TODO: process arp packet: arp request & arp reply.\n");
    struct ether_arp *arp_ether_arp;
    arp_ether_arp = (struct ether_arp *)(packet + sizeof(struct
ether_header));
    if(arp_ether_arp -> arp_op == htons(0x01)){
        arp_send_reply(iface, arp_ether_arp);
    }
    else if(arp_ether_arp -> arp_op == htons(0x02)){
        arpcache_insert(ntohl(arp_ether_arp->arp_spa),arp_ether_arp-
>arp_sha);
    }
}
```

同样需要注意网络字节序和本地字节序，需要在packet的基础上选择合适的偏移地址作为 arp的头部字段，然后对其进行分配。

接着是 arpcache.c

```
// lookup the IP->mac mapping
//
// traverse the table to find whether there is an entry with the same IP
// and mac address with the given arguments
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
```

```
    for(int i = 0; i < MAX_ARP_SIZE; i++){
        if(arpcache.entries[i].ip4 == ip4 && arpcache.entries[i].valid){
            for(int j = 0; j < ETH_ALEN; j++){
                mac[j] = arpcache.entries[i].mac[j];
            }
            return 1;
        }
    }
    return 0;
}
```

这里需要注意 arpcache表示被删除的项，需要观察其 valid 值。

```
// append the packet to arpcache
//
// Lookup in the list which stores pending packets, if there is already an
// entry with the same IP address and iface (which means the corresponding arp
// request has been sent out), just append this packet at the tail of that entry
// (the entry may contain more than one packet); otherwise, malloc a new entry
// with the given IP address and iface, append the packet, and send arp request.
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)
{
    fprintf(stderr, "TODO: append the ip address if lookup failed, and send arp
request if necessary. %x\n", ip4);
    struct arp_req *req_entry = NULL, *req_q;
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list){
        if(req_entry->ip4 == ip4){
            struct cached_pkt *temp;
            temp = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
            temp->packet = packet;
            temp->len = len;
            list_add_tail(&(temp->list),&(req_entry->cached_packets));
            return;
        }
    }

    struct arp_req *target;
    target = (struct arp_req *)malloc(sizeof(struct arp_req));
    list_add_tail(&(target -> list), &(arpcache.req_list));
    target->iface = iface;
    target->ip4 = ip4;
    target->sent =  time(NULL);
    target -> retries = 0;
    struct cached_pkt *temp;
    temp = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
    temp->packet = packet;
    temp->len = len;
    init_list_head(&(target->cached_packets));
    list_add_tail(&(temp->list),&(target->cached_packets));

    arp_send_request(iface,ip4);
}
```

这里的写法参照了上面destory的写法，主要在于对 list_for_each_entry_safe 进行调用，并对新申请的空间赋值。

```
// insert the IP->mac mapping into arpcache, if there are pending packets
// waiting for this mapping, fill the ethernet header for each of them, and send
// them out
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
    fprintf(stderr, "TODO: insert ip->mac entry, and send all the pending
packets. %x\n", ip4);
    int i;
    for(i = 0; i < MAX_ARP_SIZE; i++){
        if(arpcache.entries[i].valid == 0){
            arpcache.entries[i].ip4 = ip4;
            for(int j = 0; j < ETH_ALEN; j++){
                arpcache.entries[i].mac[j] = mac[j];
            }
            arpcache.entries[i].added = time(NULL);
            arpcache.entries[i].valid = 1;
            break;
        }
    }
    if(i == MAX_ARP_SIZE){
        arpcache.entries[0].ip4 = ip4;
        for(int j = 0; j < ETH_ALEN; j++){
            arpcache.entries[0].mac[j] = mac[j];
        }
        arpcache.entries[0].added = time(NULL);
        arpcache.entries[0].valid = 1;
    }

    struct arp_req *req_entry = NULL, *req_q;
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list){
        if(req_entry->ip4 == ip4){
            struct cached_pkt *pkt_entry = NULL, *pkt_q;
            list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry-
>cached_packets), list) {
                struct ether_header *arp_ether_header;
                arp_ether_header = (struct ether_header *)(pkt_entry->packet);
                for(int j = 0; j < ETH_ALEN; j++){
                    arp_ether_header->ether_dhost[j] = mac[j];
                }
                iface_send_packet(req_entry->iface,pkt_entry->packet,pkt_entry-
>len);
            }
            list_delete_entry(&(req_entry->list));
            free(req_entry);
        }
    }
}
```

这里主要明确 arp_req 和 cached_pkt 之间的关系

```
void *arpcache_sweep(void *arg)
{
    while (1) {
        sleep(1);
        fprintf(stderr, "TODO: sweep arpcache periodically: remove old entries,
resend arp requests .\n");
        for(int i = 0; i < MAX_ARP_SIZE; i++){
```

```
            if(arpcache.entries[i].valid == 1 && time(NULL) -
arpcache.entries[i].added > 15){
                arpcache.entries[i].valid = 0;
            }
        }
        struct arp_req *req_entry = NULL, *req_q;
        list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list){
            if(time(NULL) - req_entry->sent > 1 && req_entry->retries < 5){
                arp_send_request(req_entry->iface,req_entry->ip4);
                req_entry->retries++;
            }
            else if(req_entry-> retries == 5){
                struct cached_pkt *pkt_entry = NULL, *pkt_q;
                list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry-
>cached_packets), list) {
                    icmp_send_packet(pkt_entry->packet, pkt_entry-
>len,ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
                    list_delete_entry(&(pkt_entry->list));
                    free(pkt_entry->packet);
                    free(pkt_entry);
                }
                list_delete_entry(&(req_entry->list));
                free(req_entry);
            }
        }
    }
    return NULL;
}
```

主要是对time变量的访问，然后依照条件和注释给出的信息，发出对应的icmp包。

这次理解上的一个重点是 icmp.c

首先我们需要明确，如果收到的包是icmp的echorequest,则需要发送icmp回复，这时发送的数据包大小就是之前传入的len的大小。

而对于其它情况而言，都是ether头部 + ip 头部 + icmp 头部 + 4字节的0 + 从 ip 头部拷贝的字段 + 之后的 8 字节

所以我们面临的问题是，in_pkt 的 ip字段大小可能不是标准的 sizeof(struct iphdr),而有可能导致变长。

这对于发送 icmp 的 echoreply 报文而言，我们需要对原来的 in_pkt 计算相应的偏移，就需要用到 IP_HDR_SIZE(hdr)，但由于本地发送报文时，还是依据本地定义的结构体，所以拷贝到的对象的长度依然是标准长度。

```
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    fprintf(stderr, "TODO: malloc and send icmp packet.\n");
    const struct ether_header *icmp_ether_header0;
    icmp_ether_header0 = (const struct ether_header *)in_pkt;
    const struct iphdr *icmp_iphdr0;
    icmp_iphdr0 = (const struct iphdr*)(in_pkt + sizeof(struct ether_header));

    char *icmp_packet;

    if(type == ICMP_ECHOREPLY){
        icmp_packet = (char *)malloc(len);
```

```
        memcpy(icmp_packet + sizeof(struct ether_header) + sizeof(struct iphdr),
in_pkt + sizeof(struct ether_header) + IP_HDR_SIZE(icmp_iphdr0), len -
sizeof(struct ether_header) - IP_HDR_SIZE(icmp_iphdr0));
    }
    else{
        icmp_packet = (char *)malloc(sizeof(struct ether_header) + sizeof(struct
iphdr) + sizeof(struct icmphdr) +  IP_HDR_SIZE(icmp_iphdr0) + 8);
    }
    struct ether_header *icmp_ether_header;
    icmp_ether_header = (struct ether_header *)icmp_packet;
    struct iphdr *icmp_iphdr;
    icmp_iphdr = (struct iphdr *)(icmp_packet + sizeof(struct ether_header));
    struct icmphdr *icmp_icmphdr;
    icmp_icmphdr = (struct icmphdr *)(icmp_packet + sizeof(struct ether_header)
+ sizeof(struct iphdr));

    for(int i = 0; i < ETH_ALEN; i++){
        icmp_ether_header -> ether_dhost[i] = icmp_ether_header0 ->
ether_shost[i];
        icmp_ether_header -> ether_shost[i] = icmp_ether_header0 ->
ether_dhost[i];
    }
    icmp_ether_header-> ether_type = icmp_ether_header0 -> ether_type;

    if(type == ICMP_ECHOREPLY){
        ip_init_hdr(icmp_iphdr, ntohl(icmp_iphdr0 -> daddr),ntohl(icmp_iphdr0 ->
saddr),len - sizeof(struct ether_header),IPPROTO_ICMP);
    }
    else{
        rt_entry_t *icmp_rtable = longest_prefix_match(ntohl(icmp_iphdr0 ->
saddr));
        iface_info_t *icmp_iface = icmp_rtable -> iface;
        ip_init_hdr(icmp_iphdr, (icmp_iface->ip),ntohl(icmp_iphdr0 ->
saddr),sizeof(struct iphdr) + sizeof(struct icmphdr) + IP_HDR_SIZE(icmp_iphdr0)
+ 8,IPPROTO_ICMP);
    }
    icmp_icmphdr -> type = type;
    icmp_icmphdr -> code = code;

    if(type == ICMP_ECHOREPLY){
        icmp_icmphdr -> checksum = icmp_checksum(icmp_icmphdr, len -
sizeof(struct ether_header) - IP_HDR_SIZE(icmp_iphdr0));
        ip_send_packet(icmp_packet, len);
    }
    else{
        icmp_icmphdr -> u.um.unused = 0;
        icmp_icmphdr -> icmp_mtu = 0;
        memcpy( (icmp_icmphdr + 1), icmp_iphdr0 , IP_HDR_SIZE(icmp_iphdr0) + 8
);
        icmp_icmphdr -> checksum = icmp_checksum(icmp_icmphdr, sizeof(struct
icmphdr) + IP_HDR_SIZE(icmp_iphdr0) + 8 );
        ip_send_packet(icmp_packet, sizeof(struct ether_header) + sizeof(struct
iphdr) + sizeof(struct icmphdr) + IP_HDR_SIZE(icmp_iphdr0) + 8);
    }
}
```

还需要注意的是，icmp中定义的结构体union的大小是两个 u16,所以 4 字节的 0 就覆盖了结构体 u 所在的部分。这一部分代码主要的难点在于，搞清楚 ip包附加内容对于原先包的影响，相应的偏移量变化。

ip_base.c 部分

```
// lookup in the routing table, to find the entry with the same and longest
prefix.
// the input address is in host byte order
rt_entry_t *longest_prefix_match(u32 dst)
{
    fprintf(stderr, "TODO: longest prefix match for the packet.\n");
    rt_entry_t *req_entry = NULL, *req_q;
    rt_entry_t *temp = NULL;
    list_for_each_entry_safe(req_entry, req_q,&(rtable),list){
        if((req_entry -> dest & req_entry -> mask) == (dst & req_entry -> mask))
{
            if(temp == NULL){
                temp = req_entry;
            }
            else if((req_entry -> mask > temp -> mask)){
                temp = req_entry;
            }
        }
    }
    return temp;
}
```

这里犯了 & 的优先级和等号的错误，通过抓包以及打印调试信息，发现一直没有找到合适的最长前缀匹配，需要注意括号的问题。

```
// send IP packet
//
// Different from forwarding packet, ip_send_packet sends packet generated by
// router itself. This function is used to send ICMP packets.
void ip_send_packet(char *packet, int len)
{
    fprintf(stderr, "TODO: send ip packet.\n");

    struct ether_header *ip_ether_header;
    ip_ether_header = (struct ether_header *)packet;
    struct iphdr *ip_iphdr;
    ip_iphdr = (struct iphdr *)(packet + sizeof(struct ether_header));

    u32 ip2;
    ip2 = ntohl(ip_iphdr -> daddr);

    rt_entry_t *rtable2 = longest_prefix_match(ip2);
    ip_iphdr -> ttl--;
    if(ip_iphdr -> ttl == 0){
        icmp_send_packet(packet,len,ICMP_TIME_EXCEEDED,ICMP_EXC_TTL);
    }
    else{
        ip_iphdr -> checksum = ip_checksum(ip_iphdr);
        if(rtable2 == NULL){
            fprintf(stderr, "rtable %x\n", ip2);
            icmp_send_packet(packet,len,ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
```

```
            // void icmp_send_packet(const char *in_pkt, int len, u8 type, u8
code);
        }
        else{
            iface_info_t *port2 = rtable2 -> iface;
            for(int i = 0; i < ETH_ALEN; i++){
                ip_ether_header-> ether_shost[i] = port2 -> mac[i];
            }
            if(rtable2 -> gw != 0){
                if( arpcache_lookup(rtable2 -> gw, ip_ether_header ->
ether_dhost) ){
                    iface_send_packet(port2, packet, len);
                }
                else{
                    arpcache_append_packet(port2,rtable2 -> gw,packet,len);
                }
            }
            else{
                if( arpcache_lookup(ip2, ip_ether_header -> ether_dhost) ){
                    iface_send_packet(port2, packet, len);
                }
                else{
                    arpcache_append_packet(port2,ip2,packet,len);
                }
            }
            //arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet,
int len)
        }
    }
}
```

发送ip包这里做了大部分的工作，也是由于 iface 可以得到本地结点的ip地址和mac地址，所以在 icmp.c 的部分中，首先对于发送的包，其所有目的 ip都设置成了最终的 ip,而不是下一跳 ip,这不符合实验二 traceroute的要求。所以对于每一个包的目的ip进行修改，需要用到全局定义的变量 rtable,然后调用相应的 iface,进行发送。

这里需要明确 gw 和 iface 的 ip之间的关系，之前由于没有区分 gateway,导致在抓包时发现经常出现询问 0.0.0.0?这样地址的错误 arp 包，之后发现处于同一网段的 gw 都是 0.0.0.0，这时就需要用到 iface记录的 ip，这是容易出错的点。

```
// handle ip packet
//
// If the packet is ICMP echo request and the destination IP address is equal to
// the IP address of the iface, send ICMP echo reply; otherwise, forward the
// packet.
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    fprintf(stderr, "TODO: handle ip packet.\n");
    struct iphdr *ip_iphdr;
    ip_iphdr = (struct iphdr *)(packet + sizeof(struct ether_header));

    struct icmphdr *ip_icmphdr;
    ip_icmphdr = (struct icmphdr *)(packet + sizeof(struct ether_header) +
IP_HDR_SIZE(ip_iphdr));

    if(ip_iphdr -> protocol == IPPROTO_ICMP && ip_icmphdr -> type ==
ICMP_ECHOREQUEST){
```

```
        if(ip_iphdr-> daddr == htonl(iface-> ip)){
            icmp_send_packet(packet,len,ICMP_ECHOREPLY,ICMP_ECHOREPLY);
        }
        else{
            ip_send_packet(packet,len);
        }
    }
    else{
        ip_send_packet(packet,len);
    }
}
```

需要 include "icmp.h",这里需要做的工作比较少，就是从packet中取出 ip 头部 和 icmp 头部，然后根据 是否为 ICMP报文进行分类。